Solving Swath Problems Optimally

Russell Knight
California Institute of Technology, Jet Propulsion Laboratory
russell.knight@jpl.nasa.gov

Abstract

We present the first optimal algorithms for swath segment scheduling for orbiting spacecraft. We present a comparison between an integer program formulation and a branch and bound formulation that makes use of a flow network transformation, each capable of solving instances of these problems optimally. We also compare our techniques with the current state of the practice: the Aster scheduling algorithm. No technique strictly dominates all others, and we characterize their respective advantages and disadvantages. Note that this problem is NP-complete. The primary goal of our work is to solve the largest swath problems possible, both quickly, and where feasible, optimally..

1. Introduction

Orbiting spacecraft often have immobile imaging instruments, and generally such spacecraft maintain a fixed orientation with respect to the body that they are orbiting, therefore most instruments point straight down, or nearly straight down. In these cases, the instrument gathers data along a fixed trajectory called a swath. In practice, the swaths are broken into smaller swaths called segments. It is important to note that segments are both areas that could be imaged and intervals of time for the observation.

Figure 1 (right) shows some example segments.

The purpose of such spacecraft is to image various areas, called targets, according to the investigator's priorities. Figure 1 (left) shows an example of a target.

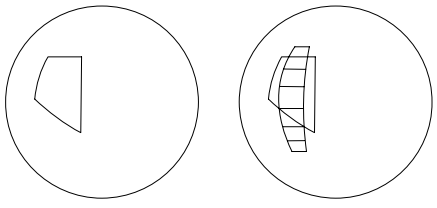

Figure 1. A target without and with segments

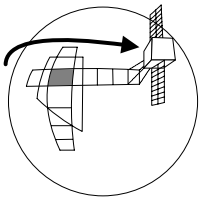

Figure 2. A pair of swaths

Overlapping segments imply a potential for waste if the overlapping area is collected and transmitted more than once. Figure 2 shows more segments as a result of a subsequent orbit by the spacecraft. (The arrow indicates the direction of travel.) Note that the segment in the center of the target (shaded area) overlaps one of the previous segments.

Collecting overlapping segments is a problem because there are limits to how many segments we may collect. This is usually due to limited on-board memory, and limited downlink times and capacities. Not surprisingly, segments are usually carefully chosen to reduce overlap. But choosing the best segments can be problematic, especially for large numbers of observations. This problem is called the *swath segment selection problem*.

2. Approach

Our approach searches the space of segment-inclusions, trying to find a set of segments that results in the most rewarding set of targets being imaged. We start with a solution that includes no segments. We use a depth first branch and bound search with a heuristic award estimator. The heuristic award estimator is a network flow transformation of the problem. The node ordering heuristic orders selections based on the reward to capacity cost ratio of a segment, given the segments that are known to be included or excluded. We refer to this technique as *Flow*.

We show the first optimal solutions for large swath segment selection problems. We also show that our technique outperforms the current state of the practice,

2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)
0-7695-2644-6/06 $20.00 © 2006 IEEE

the Aster scheduling algorithm, by approximately 39%, e.g., xxx.

Previously, we had reported on the general characteristics of the approach of using a flow network transformation of the problem as an admissible heuristic in branch-and-bound search to solve the problem quickly. See [8] for details. Here we describe the node ordering heuristics used to achieve the high level of performance reported. The heuristics are important in that they can be used not only for node-ordering in a complete search framework, but also as the greedy metric in a constructive algorithm. It is very important to realize that the context of these heuristics was to enable fast complete search, so certain decisions were made to improve the ability of these heuristics incrementally—that is, once a decision is made, the new value for the heuristic on the new, smaller, problem can be computed more efficiently than computing the same metric from scratch.

2.1. General Flow-network Formulation

To understand how the heuristics are used, it is important to understand in which context they are applied.

The goal is to generate a flow network that represents the flow of capacity usage through the problem. It is important to note that this formulation does have some limitations; most importantly it assumes that the reward for an element scales with its capacity cost, which might not be the case. Under those circumstances, the IP formulation is the more accurate, and probably should be used. The rest of this subsection describes the construction of the flow network. Given our previous example, we would first add a node called *src* and a node called *snk* to our graph (see xxx.)

Figure 3. Source and sink nodes

We then add a node for each shard, and add an edge representing the reward for collecting the shard from *src* to the elements node (see xxx). Note that capacity cost and reward must be equivalent for the network flow formulation to be used.

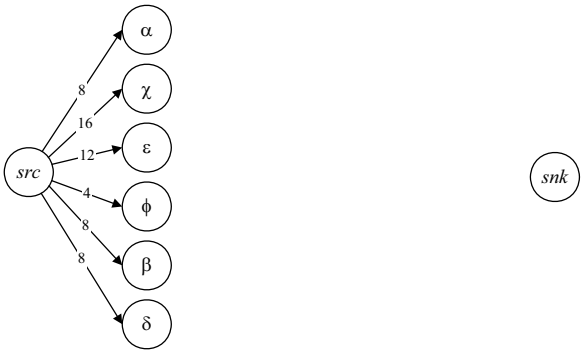

Figure 4. Shard nodes

Then, for each segment, we add a node and add an edge from each shard that the segment contains with the same capacity as the reward for the shard (see xxx). Note that in our example, shard c belongs to segment 3 and segment 9.

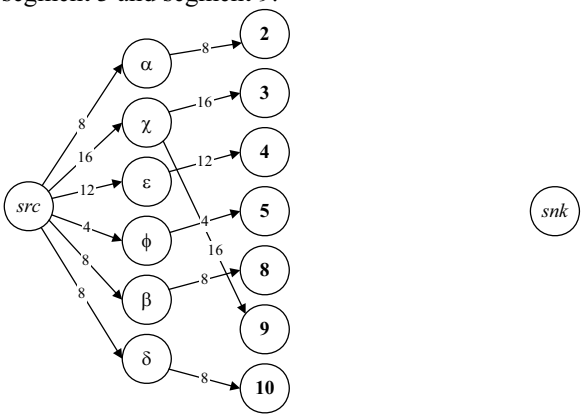

Figure 5. Segment nodes

Then, for each downlink, we add two nodes. One node collects all of the segments, and we designate it the in node. The other sends the collected reward to the sink. So, for all segments of the leg previous to the downlink, we add an edge of the same capacity as the segment from each segment to the in node. We then add an edge of the same capacity as m between the in node and the second node. Finally, we add an edge of the downlink capacity from the second node to snk. Figure 9 shows the final flow network for our example problem, where the edge-labels represent the capacity of the edge. Figure 10 shows one possible solution to the network flow problem, where the edge-labels represent the flow, and Figure xxx shows what this solution would imply when translated back into a swath segment selection problem.

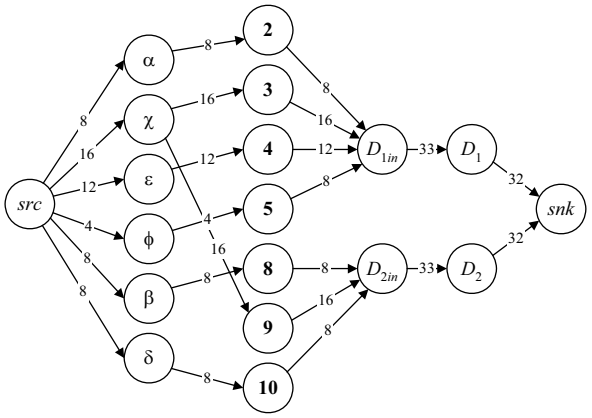

Figure 6. Complete network

We use the solution to the flow network to represent an upper-bound on the quality that we can expect for any real solution to the swath segment selection problem represented by the network.

Now we have a good heuristic reward estimator that we can apply to a traditional branch and bound search. We need a node ordering heuristic that takes a partial solution and the search options available and orders the options accordingly, hoping to find good solutions early. Specifically, we need to take the partial solution R' and consider which segments to include. We use our heuristics, described later, to decide which order in which to search the space of segment inclusions.

Search ensues thusly:

1. Let $b \leftarrow 0$, i.e., the current best quality bound
2. Let **R'** $\leftarrow \varnothing$, i.e., our current best solution
3. Let **P** $\leftarrow \varnothing$, i.e., our current partial solution
4. Let *reward*(**P**) be the summed reward of the shards collected in the partial solution **P**
5. Let *OpenList* be a priority queue of segments where priority is based on reward/cost ratio gain given the partial solution **P**. pop(*OpenList*) returns the highest valued segment while removing it from *OpenList*
6. Let *OpenList* $\leftarrow S$
7. Let *h* be a real-valued heuristic function that returns the quality of the network flow relaxation of the remaining segments in *OpenList* given the partial solution **P**
8. search(**R'**, **P**, *b*)

Recursive search routine

search(**R'**, **P**, *b*)

9. if *reward*(**P**) + *h* < *b* return, i.e., prune
10. if *OpenList* = $\varnothing$, i.e., the bottom of the recursion
11. if *reward*(**P**) > *b*
12. **R'** $\leftarrow$ **P**
13. *b* $\leftarrow$ *reward*(**P**)
14. end if
15. return
16. end if
17. *s* $\leftarrow$ pop(*OpenList*)
18. if feasible to insert *s* into **P**
19. insert *s* into **P**
20. search(**R'**, **P**, *b*)
21. remove *s* from **P**
22. end if
23. search(**R'**, **P**, *b*)
24. push *s* back onto *OpenList*

2.2. Node Ordering Heuristics

Here we describe our ordering heuristics as applied to the swath segment selection problem.

2.2.1 Detecting impossible undecided segments. If we are not modeling the carry over of extra ram between downlinks, we need only detect that the sum of the memory usage of the included segments plus the candidate undecided segment memory usage is greater than the capacity for the associated downlink. For each downlink, we keep track of a single sum called the committed cost. We also keep a list of undecided segments sorted by memory usage, least usage first. When a segment is committed and selected for inclusion or it is selected for exclusion after having been previously committed, we update the committed cost, and thereby know the residual capacity. We then index into the list according to the residual capacity. All previous segments are possible; all subsequent segments are impossible.

2.2.2 Computing the greedy incremental reward. The greedy incremental reward for a segment is the ratio of the sum of the rewards for the remaining uncollected shards over the memory usage of the segment. When an shard is collected, the reward for the segment is recalculated.

2.2.3 Calculating the computational costs of segment inclusion. Including a segment in a solution might require many computations to update the flow network. The fundamental cost of this computation lies in the discovery and reflection of depleting and augmenting paths. If all positive flow from contained shard nodes go to the segment node, then we need not compute depleting paths, and we can simply remove the nodes. Similarly, if all of the capacity reduction can be accommodated by the loss of flow of the segment plus any residual flow along the shortest path to sink, then we can simply remove the flow and update the capacities. Segment/downlink assignments that meet these criteria are trivially easy to include in the solution without changing the value of the network

flow. We simply include them, and update the flow solution and capacities. The estimate of the computational cost of segment inclusion is the number of depleting and augmenting paths that needs to be generated.

In general, we can determine if an augmenting flow of a fixed length exists without actually updating the network simply by computing the modified single-source-shortest path algorithm for a limited number of iterations. Each step in the outer loop corresponds to a node-distance in the flow. If we are interested in only local flows, then we limit this algorithm to a fixed number of steps, leading to a computation of local augmenting paths. Our technique is a specialization of this more general approach to determining the computational cost of incrementally updating a flow network.

2.2.4 Computing the best fit for an inclusion. The best fit for a segment inclusion is similar to the best fit for bin-packing. The idea is to fit as exactly as possible the memory usage of a segment with the residual capacity, using the residual capacity computation described earlier. Therefore, the best fit is the segment selection that results in the smallest residual capacity. To compute this, we use the technique of finding the impossible segments, but chose the next-to-impossible segment (i.e., the segment whose inclusion results in the least amount of residual capacity).

3. Results

We report "first solution" time and quality results for ASTER, BnBFlow, and Integer Programming for many sizes of random problems. Problems are randomly generated SSSPs.

Easily computable metrics that appear to reflect on the scale of the problems and the quality of solutions are the number of shards in H for each problem and the initial network flow approximation, thus we report these for the sizes of problems here. We report results for 100 instances per size, with a time cutoff of 1 hour. It is important to note that ASTER required less than 1 second for any instance.

Figure 7 and Figure 8 show a comparison of a solution for a typical SSSP solved using ASTER and BnBFlow. Shading indicates the solution area.

Figure 9 shows a comparison of BnBFlow (our technique using branch and bound with a network flow heuristic) and IP (Integer Programming; the "straw-man" for optimal solutions). Times are for optimal solutions, except where time is two hours. Clearly, BnBFlow took less time in finding the optimal solutions where both found optimal solutions.

Figure 10 compares BnBFlow, ASTER, and IP for solution quality. Note that we also include the network flow value as an upper bound on quality—Relaxation. This is not so important for those values where optimal values are found, but is good for comparing values where neither algorithm found optimal value, e.g., shard counts greater than 8000.

ASTER dominates in terms of generating a fast solution. But the time cost of BnBFlow is minimal compared to the solution quality. Integer programming returns an optimal solution, but does not outperform BnBFlow, and for relatively small problems doesn't terminate. Thus, in terms of any-time performance, the best strategy appears to be to first use ASTER followed immediately by BnBFlow. (See Figure 9 and Figure 10.) Note that the quality of the BnBFlow solutions continues to track the quality of the Relaxation, indicating good any-time performance by the technique.

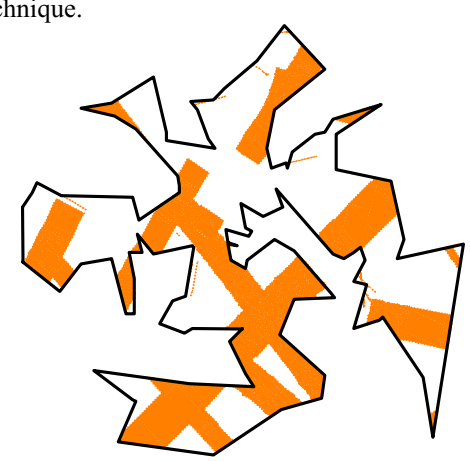

Figure 7. Aster Solution, area = 819.6

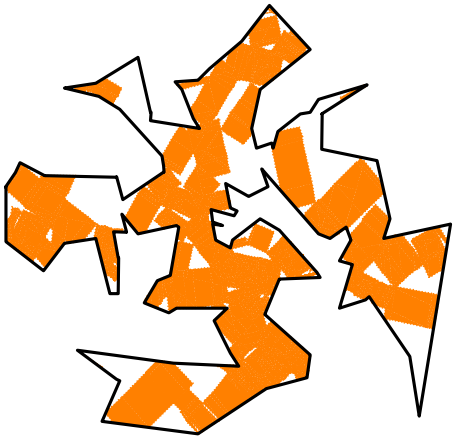

Figure 8. Flow Solution, area = 1383.29

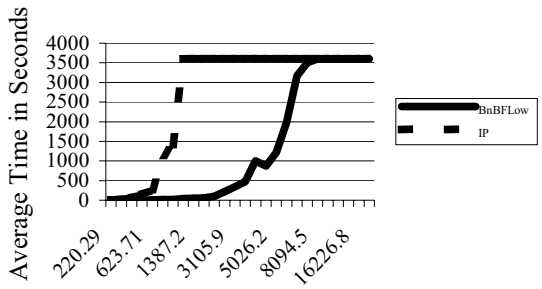

Figure 9. Solution time by problem size

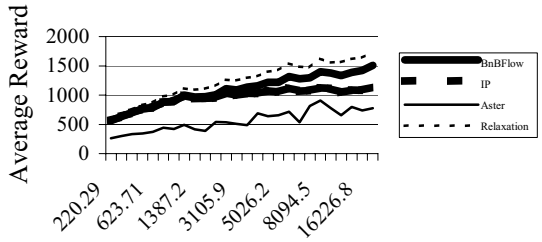

Figure 10. Solution quality by problem size

4. Related Work

The best previous problem solver for SSSPs is the ASTER system [9]. They use greedy maximization, breaking ties by choosing earliest segments first, to find a solution. Their algorithm is deterministic and very fast. Basically, they subdivide the problem into separate legs, and schedule each leg. As it turns out, a leg corresponds to a day of operations. For each leg, they include the segment that has the best reward/cost value, until no more segments can be accommodated. They break ties by choosing the earlier segment.

Work on a somewhat similar problem with more degrees of freedom is reported by [4]. In this case, a route for an aircraft-borne off zenith observatory must be planned that maintains pointing at celestial targets over an interval of time. The route is flexible (as opposed to our fixed routes) and more constraints (maximum fuel usage, round trip travel, etc.) are considered, but on-board memory is not a prohibitive factor.

For a good example of a polyhedral solution to a combinatorial optimization problem having to do with satellite scheduling (formulated as a pick-up and delivery problem), see [12].

[10] solves a constrained-memory domain with fewer types of constraints called the Mars Express Memory Dumping Problem. The system uses a portfolio approach to solving the problem as formulated in a constraint-based framework. The portfolio consists of a tabu search strategy, a random sampling strategy, and a greedy strategy.

More general constraint-based frameworks for scheduling that have been applied to spacecraft operations include that of [3], [6], and [1]. In each of these, the problem is expressed as a set of constraints to be satisfied. In the case of [3], and [6], the systems search the feasible space of domains in the constraint space. In the case of [1] the system searches both the infeasible and feasible space of value assignments, using randomized local search.

5 Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

10. References

[1] S. Chien, G. Rabideu, R. Knight, R. Sherwood, B Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. “Automating space mission operations using automated planning and scheduling.” In Proc. SpaceOps, 2000.

[2] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.

[3] J. Dungan, J. Frank, A. Jonsson, R. Morris, and D. Smith. “Advances in Planning and Scheduling of Remote Sensing Instruments for Fleets of Earth Orbiting Satellites.” Earth Science Technology Conference, 2002. Pasadena, California.

[4] J. Frank. "SOFIA's Choice: Automating the Scheduling of Airborne Observations" Proceedings of the 2d NASA Workshop on Planning and Scheduling for Space, March 2000.

[5] M.R. Garey and D.S. Johnson. Computers and Intractability. W.H. Freeman and Company, San Francisco, 1979.

[6] M. Ghallab and H. Laruelle, “Representation and control in IxTeT, a temporal planner”, in Proceedings of

the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94), pp. 61-67, Chicago, IL, (1994). AAAI Press, Menlo Park.

[7] R. M. Karp "Reducibility among combinatorial problems." In R. E. Miller and J. W. Thatcher (eds.) Complexity of Computer Computations, Plenum Press, New York, 85-103.

[8] R. Knight and B. Smith. "Optimally Solving Nadir Observation Scheduling Problems." International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS 2005). Munich, Germany. September 2005.

[9] H. Muraoka, R. H. Cohen, T. Ohno, and N.Doi. "ASTER Observation Scheduling Algorithm." SpaceOps 98. 1998, 1-5 June, Tokyo, Japan.

[10] A. Oddi, N. Policella, A. Cesta and G. Cortellessa. "Generating High Quality Schedules for Spacecraft Memory Downlink Problems." Ninth International Conference on Principles and Practice of Constraint Programming, 29 September - 3 October, 2003, Kinsale, County Cork, Ireland.

[11] C. H. Papadimitriou and K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[12] K. Ruland. Polyhedral solution to the pickup and delivery problem. Washington University, Sever Institute of Systems Science and Mathematics. http://rodin.wustl.edu/~kevin/dissert/dissert.html (Dissertation). St. Louis Missouri, 1995.

[13] A. Schrijver. Theory of Linear and Integer Programming, Wiley, 1986.

[14] W. Zhang. "Depth-First Branch-and-Bound versus Local Search: A Case Study." In Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000), pages. 930-935, Austin, Texas, 2000.